NASA Technical Memorandum 104098

A Case Study for the Real-Time Experimental Evaluation of the VIPER Microprocessor

Victor A. Carreño
Rob K. Angellatta

September 1991

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

A Case Study for the Real-Time Experimental Evaluation of the VIPER Microprocessor

Victor A. Carreño and Rob K. Angellatta

Introduction

This paper describes an experiment to evaluate the applicability of the VIPER (Verifiable Integrated Processor for Enhanced Reliability) microprocessor to real-time control. The VIPER microprocessor was invented by the Royal Signals and Radar Establishment (RSRE), U.K., and is an example of the use of formal mathematical methods for developing digital electronic systems with a high degree of assurance on the system design and implementation correctness. The design of the VIPER microprocessor was guided by several criteria and restricted by engineering and verification methods including:

- Developing a microprocessor for use in safety-critical applications [4].
- Architectural complexity restrictions imposed by limitations of the methods used to verify correctness of design and implementation [5].
- Instruction set and architectural characteristics that will 'encourage' the production of safe software programs and hardware designs.
- Creating gate level descriptions of the microprocessor for verification as well as silicon fabrication.

The simplicity of the VIPER instruction set as well as its architectural design has resulted in throughput performance penalties. Thus, an experiment was conducted to assess the VIPER characteristics when used to perform real-time control law computations. A description of the VIPER microprocessor is presented in the next section.

The experiment consisted of selecting a control law, writing the control law algorithm for the VIPER processor and providing real-time, dynamic inputs to the processor and monitoring the outputs. The control law selected and coded for the VIPER processor was the yaw damper function of an automatic landing program for a 737 aircraft. A dynamic model of a 737 aircraft, running on a Digital Equipment Corporations (DEC) VAX-11/750 (VMS) computer, was then controlled by the automatic landing program of which the yaw damper portion was running on the VIPER Single Board Computer (SBC).

The mechanisms for interfacing the VIPER SBC to the VAX host are described in the Interface section. The yaw damper control law as well as the Vista programming language for the VIPER are described in the section Control Law. The section Results presents run time experiences, performance evaluation, and comparison of VIPER and FORTRAN yaw damper algorithm output for accuracy estimation.

The VIPER Microprocessor and Single Board Computer

VIPER

The VIPER microprocessor was developed as part of a program to "...devise means of writing formal specifications for safety and security critical hardware and then prove that a particular realization of this specification conforms with the top level requirement." The VIPER top level specification was written in LCF-LSM (Logic of Computable Functions-Logic of Sequential Machines)[8]. At the lower, electronic block model level the VIPER was described in both LCF-LSM and ELLA [10], the latter description used to interface with VLSI CAD tools and to provide for modeling of some physical characteristics.

The VIPER major state machine description was formally verified against the top level specification using the HOL [9] mechanized theorem prover. This work is described in reference [2]. An electronic block model description was generated in the HOL system logic and many properties of the block model were verified as described in reference [3]. Extensive simulation of the VIPER top level specification was also performed by RSRE using an 'animation' of the specification written in Algol68 and ML [6,13]. Simulations of the major state machine and block model were also performed using Algol68 and ELLA respectively [5].

The VIPER is a 32-bit processor with a 20-bit memory address space and 20-bit peripheral space. A unique feature of the processor is a STOP state were the processor halts operation due to an illegal input or internal condition. Some of these conditions will be discussed in this section. The processor has 3 general purpose 32-bit registers, A, X and Y; a 20-bit program counter register P; and a one bit flag register B. Figure 1 is a block diagram of the VIPER architecture. Instructions are single 32-bit words. The 32-bit instruction word is divided into two fields, a 12-bit function (opcode) field and a 20-bit address/data field. The addressing modes supported by the processor are immediate, direct, and indexed addressing. Arithmetic and logical instructions, with the exception of shift, are of the form D := R op M, where D is the destination register, R is a 32-bit operand from a register, and M is a 32-bit operand from memory or a 20-bit operand from the address/data field of the instruction. Destination D can be any of the registers A, X, Y or the program counter P. Since the program counter is 20-bit long, attempting to load a binary number greater than 20 bits (i.e. a *1* in any of the higher 12-bits) will cause a STOP condition. The operand R can be registers A, X, Y or program counter P. When operand M is the 20-bit address/data field or operand R is the program counter P, the operand will be 'padded' with zeros for the top 12 bits.

The 12-bit opcode field is divided into five subfield for instruction decoding. The opcode subfields are:

- 2-bit source register field
- 2-bit memory field
- 3-bit destination field
- 1-bit comparison field
- 4-bit function field

Table 1 shows the operations for the 4-bit function field. The following instructions in the table will cause the processor to STOP for the given condition:

Y := P then P := M	stop if $M > 2^{20} - 1$
D := R + M	stop on overflow
D := R - M	stop on overflow
D := R * 2	stop on overflow
spare instruction	stop if encountered

A complete description of the instruction decoding format as well as other conditions that will cause the processor to STOP can be found in reference [10].

Single Board Computer

The VIPER SBC was designed and fabricated at RSRE for evaluation purposes. One SBC assessment board was made available to NASA Langley Research Center (LaRC) under a cooperative agreement with the U.K. Ministry of Defense (MOD). The main components of the SBC used in this experiment are clock generating circuitry, a VIPER microprocessor, an Intel 8039 microprocessor, 8k bytes of 32-bit VIPER RAM, 32k bytes of 8-bit PROM where a VIPER test program resides, and 4k bytes of 8-bit PROM for the 8039 processor. A block diagram of the SBC is depicted in figure 2. The function of the 8039 processor is to control the operation of the VIPER processor, to load the VIPER RAM and to communicate with a terminal or host computer via an RS232 port. The 8039, interpreting a series of instructions invoked from the terminal or host computer, will allow the experimenter to:

- load the PROM test program into the VIPER RAM
- load a VIPER instruction into any arbitrary RAM location
- load a file, directly by typing on the terminal, or from a host computer file system
- step the VIPER clock n half clock cycles (for n = 1,2,3 ...)
- execute n VIPER instructions (n = 1,2,3 ...)

- execute until program counter contains address x, where x is specified by the user
- let the VIPER clock run freely

Other functions are possible and are described in detail in reference [1].

Experiment

The yaw damper control law algorithm of a 737 autoland program was selected for the experiment. The yaw damper was coded for the VIPER SBC and is described in the section Control Law. A block diagram representation of the software organization of the experiment is illustrated in figure 3. In this figure the boxes represent software code and the interconnection lines are as follows:

UNIBUS The UNIBUS subsystem provides for communications between I/O devices and the VAX-11/750 processor. The asynchronous, bidirectional UNIBUS is the standard interconnect for attaching low and medium speed peripheral devices to the VAX-11 family of computer systems.

DSS The DSS11 peripheral provides UNIBUS compatible computer systems with digital input capabilities. Manufactured by Digital Equipment Corporation, the DSS11 module consists of 49 optically isolated inputs, which includes 48 nonbuffered sense-data inputs and one interrupt input.

DRS The DSS companion peripheral, the DEC DRS11 provides UNIBUS compatible computer systems with digitial output capabilities. The DRS11 module consists of 48 open-collector buffered outputs plus one interrupt input.

Interface Board Designed and Fabricated at LaRC to interface the DRS and DSS to the VIPER SBC through two 64 pin connectors on the SBC. A description of the Interface Board is in section Interface.

Subroutine Call Calls from the main body of the autoland to functions and procedures. For the purposes of the experiment, the code implementing the yaw control system resides on the VIPER SBC. The Subroutine Call to yaw damper represents a subroutine call to the module responsible for communicating with the VIPER SBC.

Device Driver A specific component of the VMS operating system software, the device driver is responsible for all communication between the application software and the specified hardware – the DRS and DSS peripherals. The Device Driver Call, as depicted in the diagram, represents the application software's interaction with the VMS operating system software.

Sensor Inputs The sensor inputs are physical parameters generated by the 737 simulator according to flight conditions. They are used as the input to the autoland control. Sensor inputs include air speed and rudder deflection.

Actuator Control Outputs The autoland generates output signals that control surface actuators. The surface deflections resulting from the autoland commands together with external conditions are used by the 737 simulator in a closed loop to model flight conditions.

The autoland program is a linear control system that runs in real time simultaneously with the aircraft simulation. It is coded in FORTRAN and was slightly modified for the experiment. When the yaw damper subroutine of the program is invoked, communication is established with the SBC. The autoland program will pass four values to the VIPER yaw damper and wait for the computation result. If the result is not computed within a preset time-out period, autoland execution will be aborted and an error flag raised in the host machine.

A double precision FORTRAN version of the yaw damper was also coded for accuracy measurements. Different autoland configurations can be used during experiment runs where any or all of the yaw damper algorithms are executed. This allows for comparison of the computed values for all inputs during simulated landings.

The simulated landings start at four different, user selectable, altitudes and horizontal locations. The user also selects wind direction, wind velocity, and gust amplitude. For this experiment, wind direction was varied from zero degrees (head-on wind) to 90 degrees (cross wind) in reference to the runway orientation. Wind velocity was varied from zero knots to the maximum wind which will not exceed landing restrictions. Maximum wind landing restrictions were taken to be 40 knots head wind and 15 knots cross winds. Therefore, for wind direction angles (with respect to the runway) between zero and 20.56 degrees, wind speed was varied between 0 and $40/\cos\theta$ knots. For angles between 20.56 and 90 degrees, speed was varied between 0 and $15/\sin\theta$ knots. The wind gust amplitude was set to 5/3 of the constant wind velocity.

For each of the three routines implementing the yaw damper algorithm, single precision FORTRAN, double precision FORTRAN, and VIPER, 3600 simulated landings were performed. Each landing lasts approximately 3 minutes and the yaw damper subroutine is invoked 20 times per second.

Interface

A communications interface between the host VAX-11/750 and the VIPER SBC was developed employing a set of DEC DSS11/DRS11 48 channel digital I/O modules and a custom designed interface board. This communication arrangement provides for fast parallel exchange of data in real time and was used as the most convenient way of implementing the experiment. The communication link was not intended to be a high integrity design.

The DSS11 module is a UNIBUS peripheral furnishing the VAX with digital input capabilities; specifically, 48 nonbuffered sense-data inputs and a single interrupt input are available with each DSS11 module. A single DSS11 module residing on the VAX provides an input path from the interface board. Data is transferred from the interface board to the VAX through two registers fashioned from 40 of the 48 input channels of the DSS11 module. As depicted in figure 4, the *result register* provides a 32-bit data register

for transferring application data. The *status register* provides an 8-bit data register for transferring status information to the VAX. Additionally, the interface board signals the VAX when input data is available through the DSS11 interrupt input.

The DRS11 module is a UNIBUS peripheral giving the VAX digital output capabilities. Each DRS11 module consists of 48 open-collector buffered outputs and a single interrupt input. A single DRS11 module connected to the VAX comprises the output path to the interface board. Similar to the VAX input path, the VAX output to the interface board consists of two registers fashioned from the DRS11 module. A 32-bit wide *data register* handles the application data transfer and an 8-bit wide *identification register* provides additional control information.

The interface board, IB, was designed specifically to perform a data exchange without hand shaking or data acknowledgement. This was necessary since the DSS and DRS boards do not contain any control lines apart from an interrupt line. Figure 4 shows 40 data lines going from the DRS to the IB, 41 lines going from the IB to the DSS, 40 data lines and an interrupt line, 32 bidirectional data lines from the IB to the VIPER SBC, and 8 control/address lines from the VIPER SBC to the IB. The timing diagrams for the IB are in figures 5a and 5b. During a read cycle the IB will respond in a maximum of 77 ns which is within the 150 ns requirement of the VIPER short read cycle as shown in figure 5a. During a write cycle, figure 5b, the IB data (DATAI) are stable at the time of the latch clock. Figure 6 is the IB schematic.

Five data values are exchanged between the VIPER SBC and the VAX. Four values (the input for the yaw damper) are requested by the VIPER SBC and one value (the result) is sent from the VIPER SBC to the VAX after computation has taken place. The VIPER SBC input request is performed by writing a request code to the status register and simultaneously issuing an interrupt. The VIPER SBC will then poll the 8-bit data identification register until the data requested appears on the register. The VAX will service the interrupt by reading, through the DSS board, 40 bits from the result and status registers and checking the 8 most significant bits (the status register). The VAX will then write to the DRS the data requested.

Since this sequence is completely asynchronous, the data in the data input and data identification registers could be latched when the lines are unstable, i.e. when the DRS is updating the values, as in figures 7b and 7c. The two possible cases are shown in these figures. First, the expected value is in the identification register (figure 7b). However, the data may or may not be valid in the data input register. Therefore, a second latch is performed on the data input and identification registers. If the expected value is in the identification register a second time, the data input register is read and it's content taken as good data. If the expected value is not in the identification register a second time, an error handler is invoked. In the second case (figure 7c) the expected value is not in the identification register and polling continues. Two more latches will be performed thereafter. As in the first case, if the expected value is not present two consecutive times, an error handler will be invoked. During the experiments, no error handler calls were issued. In figure 7a all latching occurs while the input data is stable. A flow chart of the polling algorithm is shown in figure 8. The VIPER SBC sends the result of the

computation by writing it to the result register and then writing a result code on the status register and simultaneously issuing an interrupt (interrupts are issued every time the status register is written to). The VAX will service the interrupt by reading the result and status registers as above. Since a result code exists in the status register, the 32 least significant bits will be used as the result from the VIPER yaw damper.

Control Law

Vista

The control algorithm was written using the Vista programming language. Vista is described as a high-level assembly language [11]. It was designed including only constructs that will allow effective static code analysis. Recursion, GOTO statements, and pointers are not possible in Vista. Vista instructions correspond, in most cases, one to one with VIPER machine instructions. Compound instructions can be clearly decomposed into their single constituencies as in the following example. The instruction

```
WHILE (A:= var1; A := A + var2; var3 := X) A < var3
DO
. . .
OD;
```

is a sequence of instructions as follows:

```
A := var1;
A := A + var2;
var3 :=X;
TEST A < var3;
JUMP IF;
. . .
JUMP;
```

The actual location where the target program will be placed in memory is determined by the programmer using region declarations. Four types of regions are possible: CODE, DATA, CONST, PERI. There can be more than one region of each type, as demonstrated in the following example declaration.

```
CODE   main_program      FROM    0  TO  511;
CODE   first_procedures  FROM 4096  TO 5119;
CODE   second_procedures FROM 5120  TO 6143;
DATA   global            FROM  512  TO 1023;
CONST  fix               FROM 1024  TO 2047;
PERI   io                FROM  112  TO  114;
```

The directive CODE IN is used before code to change the selected region. For example

```
CODE IN second_procedures
PROC lucy:
...
END;
PROC george:
...
END;
CODE IN first_procedures
PROC sarah:
...
END;
```

will place procedures lucy and george in region second_procedures and procedure sarah in region first_procedures.

Variables have to be declared before they are used in the program or procedure. Three types of variables can be declared: INT, a 32-bit integer number; BITS, essentially the same as INT; and CHAN, a variable that can only be used with instructions INPUT and OUTPUT and defines an input/output address. DATA and CONST regions are used for variables and constants respectively. The region PERI will assign the address space for input/output. In the following sequence, light will be associated with I/O address 8 and speed with memory address 2048. The sequence will produce output of register X to I/O address 8, register A to I/O address 9, and register A to I/O address 1024.

```
DATA      global              FROM  2048  TO  3071;
PERI      actuators           FROM     8  TO    16;
PERI      terminal            FROM  1024  TO  1025;
PERI IN   actuators;
INT       speed;
CHAN      light,water_valve;
...
OUTPUT    X, light;
OUTPUT    A, water_valve;
PERI IN   terminal;
CHAN      screen;
...
OUTPUT    A, screen;
```

Yaw Damper

The yaw damper control is a third order control system with three integrators, a clamping function, and a non-linear interpolation function implemented by a look-up table. The s-plane transfer function of yaw damper is shown in figure 9. Integration is performed using

a trapezoidal numerical approximation. The two inputs to the yaw damper are: CAS, calibrated air speed in knots; and RBDEG, rate of change of yaw in degrees per second (yaw rate). Two additional inputs are necessary in the computational implementation: RESET and T(time). The output is DELRYD, rudder deflection due to yaw damper. The yaw damper output is combined with the rudder command, DELRC, to produce the actual rudder deflection, DELR. The yaw damper output is limited to -4 to +4 degrees, and the total rudder deflection to -24.999 to 24.999 degrees.

The implementation algorithm for the VIPER yaw damper was written using scaled integer arithmetic and special purpose multiplication and division procedures. Multiplication and division are not implemented in hardware on the VIPER microprocessor and Vista-VIPER does not provide floating point arithmetic. The multiplication procedure, SMULT (scaled multiplication), delivers the product of the two factors divided by 2^{31}: SMULT(A,X) = $\frac{A*X}{2^{31}}$. For the implementation in this experiment, the factor X was always taken as positive and A could be positive or negative in two's complement representation. Multiplication is implemented by partial product additions and right shifts. The 31 least significant bits have been truncated at the end of multiplication from a 63-bit product. This method allows full usage of the 32 bits to represent a number. Division is implemented in the inverse way by: SDIV(A,X) = $\frac{X}{A} * 2^{31}$. In this implementation A and X could be positive or negative in two's complement representation. Since the largest positive number in two's complement (using 32 bits) is $2^{31} - 1$, when $|A| \leq |X|$ the quotient will be $2^{31} - 1$ or 2^{31} for positive or negative quotients respectively. If division by zero is performed (A=0) the same result as $|A| \leq |X|$ will be obtained.

Scaling of variables is static. It was performed by analyzing each variable and determining it's maximum range. The scale factor is 2^{31} divided by the smallest power of two which is larger than the maximum variable value. For example variable CAS was determined to have a range of 0 to 1000 knots. The smallest power of two larger than 1000 is $1024 = 2^{10}$ thus the scale factor is 2^{21}. Table 2 shows variable maximum ranges and scale factors. Variables in the table correspond to variables on figure 8. The Vista and FORTRAN yaw damper algorithms are in appendix A and B respectively.

Results

During preliminary testing of procedures, three software errors caused the VIPER to enter the STOP state. Two of the errors were in the multiplication procedure and due to overflow. The first instance of overflow was due to a mathematical shift-right being used where a logical shift should have been used. In a mathematical shift-right the sign bit (MSB) is copied and when absolute binary numbers are used it does not implement the desired division by two function. The second error was introduced when the mathematical shift was replaced by a logical shift through register B. Register B was altered by the logical shift instruction and therefore not set properly for the next instruction as before the change. The third error was in the main program and all procedures. It was discovered when testing procedure ONED and was due to incorrect use of array indexes. When an array VECT[n] is declared, valid index range is 0 to n-1. The range 1 to n was incorrectly

used causing the error. During more than 3600 simulated landing runs the VIPER did not enter the STOP state.

A synchronization problem was also encountered between the VIPER SBC and the VAX host. It appears that the VAX host was losing some of the interrupt signals which caused both VIPER and the VAX host to wait for each other with a subsequent time out. Checks with a digital analyzer confirmed that interrupts were been issued by the interface board (IB) and lost between the DSS and the VAX. The condition was observed approximately every 4 landings or 72,000 interrupts. After the VAX host was removed from the local DECNET network and all nonessential processes shut-down, the lost interrupts were significantly reduced to approximately 1 every 2,000,000 interrupts. The testing proceeded with this configuration.

Timing of the VIPER yaw damper algorithm revealed that, excluding inputs and outputs, it took approximately 5.5 ms for total execution. 4.7 ms were used by the multiplication procedure and 0.6 ms by the division procedure. Ten multiplications and two divisions were performed per call. Note that for this implementation of the yaw damper function, excluding inputs and outputs, 96.36 percent of the total execution time was spent performing arithmetic operations. It is very likely that all procedures could be optimized to improve performance.

The accuracy of the VIPER and FORTRAN yaw dampers was checked against a double precision floating point FORTRAN version. The accuracy was measured in terms of maximum relative error, maximum absolute error, average relative error, and average absolute error. This parameters were calculated as follows:

$$maximum\_relative\_error = MAX_{i=1}^{n}(ABS\left(\frac{VIPorFORout_i - Ref_i}{Ref_i}\right))$$

$$maximum\_absolute\_error = MAX_{i=1}^{n}(ABS(VIPorFORout_i - Ref_i))$$

$$average\_relative\_error = \sum_{i=1}^{n} ABS\left(\frac{\frac{VIPorFORout_i - Ref_i}{Ref_i}}{n}\right)$$

$$average\_absolute\_error = \sum_{i=1}^{n} ABS\left(\frac{VIPorFORout_i - Ref_i}{n}\right)$$

where

MAX is the largest number out of n numbers in a single landing run,

ABS is the absolute value,

VIPorFORout is the VIPER or FORTRAN yaw damper output value, and

Ref is the double precision reference value.

In three cases out of 3600 the VIPER maximum relative error was greater than the FORTRAN version. In all cases the VIPER maximum absolute and the average relative and absolute errors were smaller than the FORTRAN version as compared with the double precision yaw damper. The average relative VIPER error was typically half that of the FORTRAN. This demonstrates that an accuracy comparable with a floating point algorithm can be achieved with an integer number algorithm by using scaling methods. An example result for a landing run is given in table 3.

Conclusions

The VIPER microprocessor performance is in the 0.5 to 0.7 million instructions per second (native MIPS) range. The lack of built-in multiplication significantly decreases the effective performance of the processor. The VIPER processor is suitable for embedded applications as demonstrated in this experiment. Approximately 10 subroutines, equivalent to the yaw damper, could be run in real time by a VIPER processor.

Since floating point representation is not provided in the Vista-VIPER environment, variable scaling is necessary in most control applications. Detailed knowledge of the control algorithm allows the programmer to fully exploit the 32-bit integer representation. Intermediate arithmetic results create an extra burden on the programmer as he/she must ensure that no overflow occurs. An accuracy penalty must be paid if the maximum value range resulting from an arithmetic operation is not known. Also, the ordering of arithmetic operations must be considered.

Based on previous experiments conducted at AIRLAB on redundant, reconfigurable systems it was found that the redundancy management takes a considerably large amount of CPU time. For instance, the Software Implemented Fault-Tolerace (SIFT) computer's operating system overhead consumes a minimum of 50 percent of the frame size [12]. The SIFT CPUs, the Bendix BDX 930, has a performance of approximately 0.9 MIPS [7], slightly greater performance than the VIPER SBC. Therefore, it is the opinion of the writers that the VIPER processor, having a limited performance capability, would be inadequate for redundant multichannel systems where redundancy management (including fault detection, isolation, and reconfiguration) and task scheduling are done dynamically. The overhead associated with these tasks will consume most of the VIPER processing capability. Strategies for dynamic task scheduling and many aspects of redundancy management are, at the time of writing, not formally verifiable for real-time systems; thus, these features are not recommended for use in safety critical systems. Systems which employ dedicated circuitry for multichannel functions will be more amenable for VIPER use.

References

[1] Bradford, P. J., Kershaw, R. J. W., Moss, P. *RSRE VIPER Assessment Board.* RSRE Divisional Memo CC2 405-86, June 1986.

[2] Cohn, A. J., *A Proof of Correctness of the VIPER Microprocessor: The First Level.* University of Cambridge Computer Laboratory, Technical Report 104, January 1987.

[3] Cohn, A. J., *Correctness Properties of the VIPER Block Model: The Second Level.* University of Cambridge Computer Laboratory, Technical Report 134, May 1988.

[4] Cullyer, W. J., *VIPER Microprocessor: Formal Specification.* RSRE Report 85013, October 1985.

[5] Cullyer, W. J., *Implementing Safety-Critical Systems: The VIPER Microprocessor.* RSRE Divisional Memo CC2 411-87, August 1987.

[6] Cullyer, W. J., *VIPER: Simulation of Specification - User's Guide.* RSRE Memorandum 3972, September 1986.

[7] Goldberg, J., et al., *Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer.* NASA Contractor Report 172146, February 1984.

[8] Gordon, M. J. C., *LCF-LSM, A System for Specifying and Verifying Hardware.* University of Cambridge Computer Laboratory, Technical Report 41, September 1983.

[9] Gordon, M. J. C., *HOL: A Machine Oriented Formulation of Higher Order Logic.* University of Cambridge Computer Laboratory, Technical Report 68, 1985.

[10] Kershaw, R. J. W., *VIPER: A Microprocessor for Safety-Critical Applications.* RSRE Memorandum 3754, December 1984.

[11] Kershaw, R. J. W., *Vista User's Guide.* RSRE Divisional Memo CC2 401-86, April 1986.

[12] Palumbo, D. L., Butler, R. W., *Measurement of SIFT Operating System Overhead.* NASA Technical Memorandum 86322, April 1985.

[13] Wikström, Å., *Functional Programing Using Standard ML.* Prentice Hall, 1987.

Appendix A: Vista-VIPER Yaw Damper Program

the Vista program

```
-- Program to run a yaw damper control law in the VIPER SBC using data
-- values from the VAX 737 autoland. The computed value is DELRYD
-- ruder deflection due to yaw damper and is writen to the VAX through the
-- VIPER VAX interface board.
--
-- written by:  Victor A. Carreno
-- vac@air12.larc.nasa.gov
-- date: March 30,1990
-- last modified Aug 13, 1990
--

PROGRAM yaw damper:

CODE main   FROM    0 TO  511 ;
CODE proc   FROM  528 TO 3071 ;
DATA local  FROM 3100 TO 3200 ;
DATA global FROM 3202 TO 3400 ;
CONST fix   FROM 4064 TO 4095 ;
PERI io     FROM    0 TO   15 ;

DATA IN global;
CODE IN main;

INT var[5], svar[5];  -- var[0] = not used, var[1] = CAS,
-- var[2] = RBDEG, var[3] = RESET, var[4] = T
INT ys5; -- control loop state

INT state[4]; -- integrators states

INT o1index; -- interpolate procedure state
INT xst[9]; INT fst[9]; -- value array

CHAN dummy1,ireg0,ireg1,dummy2; -- dummy1 -> 0000, ireg0 -> 0001, ireg1 -> 0010
CHAN ireg2,dummy3[3],ireg3; -- dummy2 -> 0011, ireg2 -> 0100, ireg3 -> 1000
Y := 0;
ys5 := Y; state[1] := Y; -- initialize control loop and integrator states
state[2] := Y; state[3] := Y; -- this is not necesary if the first call to
-- yaw damper is in reset mode.
```

```
Y := 1;
o1index := Y;

-- table for PROC oned; interpolation procedure
-- input  output
--  CAS   KPSDOT
A := 0; X := 2147483648; --   0.0, 1
xst[1] := A; fst[1] := X;
A :=  209715200; X := 2147483648; -- 100.0, 1
xst[2] := A; fst[2] := X;
A :=  256691405; X := 1642824991; -- 122.4, 0.765
xst[3] := A; fst[3] := X;
A :=  316250522; X := 1309965025; -- 150.8, 0.610
xst[4] := A; fst[4] := X;
A :=  432013312; X := 1073741824; -- 206.0, 0.500
xst[5] := A; fst[5] := X;
A :=  612368384; X :=  848256041; -- 292.0, 0.395
xst[6] := A; fst[6] := X;
A :=  943718400; X :=  665719931; -- 450.0, 0.310
xst[7] := A; fst[7] := X;
A := 2097152000; X :=  665719931; --1000.0, 0.310
xst[8] := A; fst[8] := X;

WHILE TRUE
DO
CALL get; -- input CAS, RBDEG, RESET, T
-- CALL scale; -- convert floating to int and scale
CALL control; -- execute control law
CALL put; -- output DELRYD
CALL delay -- delay XXXus before next input

OD;
STOP; -- end yaw damper


-- ******* FIRST LEVEL PROCEDURES *******
-- ******* called from the main program


CODE IN proc;
DATA IN local;

PROC get: -- get values from interface board
```

```
-- ( inputs, none )
-- ( outputs, var[1..4] )
-- ( calls, error with A = 1 )
-- ( destroys, A, X )
BEGIN
INT vtemp,index;
X := 1; -- read values 1..4 using X as index starting with one
WHILE X < 5
DO
OUTPUT X, ireg1; -- output to ireg1 will issue an interrupt.
index := X; -- request value(X)

WHILE (A := INPUT ireg3; A := A AND 16rFF) A /= index

DO -- Read until requested
-- value is in registers.
  SKIP -- ireg3 is label of data in ireg2.
OD; -- reg3 is 8 bits long so is ANDed with 000000FF

vtemp := A; -- when requested value is in registers, save
A := INPUT ireg3; -- label and read second time to avoid
A := A AND 16rff; -- unstable data error.
IF A = vtemp -- check second value
THEN  A := INPUT ireg2; -- good data
var[X] := A -- load value

ELSE A := 1; CALL error
FI;
X := X + 1; -- increment index
Y := 16r800; -- Delay to avoid VAX missing
WHILE (Y := Y - 1) Y /= 0 -- next interrupt (approx. Xms).
DO SKIP
OD

OD

END; -- end get



PROC scale:

BEGIN
SKIP
```

```
END; -- end scale


PROC control:

BEGIN


INT nfour = -536870911; -- -4 scale 1 = 2^27
INT four =   536870911; -- 4 scale 1 = 2^27
INT zp955 = 2050846884; -- 0.955 scale 1 = 2^31
INT zp57  = 1228092215; -- 0.571895 scale 1 = 2^31
INT zp84  = 1803886264; -- 0.84     scale 1 = 2^31

INT ys1, ys3, ys7;

IF (Y := var[3]) Y /= 0 -- if reset then
THEN Y := 0; ys5 := Y FI; -- i.c. YS5 = 0


A := var[1]; -- scaled CAS; 1 =2^21

CALL oned; -- CAS in A, KPSDOT in A on return
X := A;  -- put KPSDOT in X; 1 = 2^31 = 2.147483648x10^9
-- KPSDOT allways positive <= 1

A := var[2]; -- scaled RBDEG; 1 = 2^26

CALL smult; -- VYS1 = RBDEG * KSDOT; 1 = 2^26
X := zp84;
CALL smult; -- YS1 = VYS1 * .84; 1 = 2^26
A := A ADD 0; -- clear B
A := A<<1; -- scale YS1 to 1 = 2^27
X := 1; -- first order lag #1
CALL folag; -- YS2 = folag1(YS1); 1 = 2^29
X := 1; -- w out #1
CALL wout; -- YS3 = wout(YS2); 1 = 2^30
ys3 := A; -- save YS3
X := zp955; -- scale 1 = 2^31
A := ys5; -- 1 = 2^29
CALL smult; -- A := YS6 = YS5 * 0.955; 1 = 2^29
X := 2; -- first order lag #2
```

```
CALL folag;  -- A := YS7 = folag2(YS6); 1 = 2^31
A := A/2; -- scale YS7 to 2^30 for addition
ys7 := A; -- 1 = 2^30

X := ys3; -- 1 = 2^30

Y := var[4];  -- Y := T
IF Y = 0 THEN ys7 := X FI;  -- if Time = 0 then YS7 := YS3
A := X - ys7; -- YS5P1 = YS3 - YS7; 1 = 2^30
A := A ADD 0; -- clear B
A := A<<1; -- scale YS5P1 to 2^31
X := zp57; -- 9.15 scale 1 = 2^27
CALL smult; -- YS5P2 = YS5P1*9.15; 1 = 2^27

IF A < nfour THEN A := nfour FI; -- make nfour & four 4*2^27-1
-- to avoid overflow to bit 32
-- making a positive number negative
-- on the YS5 scaling below.
-- when YS5 = 4 then A = 2^31-1
IF A > four THEN A := four FI; -- YS5 = -4 < YS5P2 < 4
A := A ADD 0; A := A<<1;
A := A ADD 0; A := A<<1; -- scale YS5 to 1 = 2^29
ys5 := A -- update YS5

END; -- end control

PROC put: -- send value in A to interface board
BEGIN

X := 16rff;
OUTPUT A, ireg0;
OUTPUT X, ireg1 -- genarate interrupt with X in control reg

END; -- end put

PROC delay:
BEGIN
Y := 16r1000; -- approx Xms.
WHILE (Y := Y - 1) Y /= 0
DO SKIP
OD

END; -- end delay
```

```
-- ****** SECOND LEVEL PROCEDURES ****** --
-- ****** called from other procedures

PROC oned:

-- input in A CAS
-- output in A KPSDOT

BEGIN

INT driver,sda,sdx; -- local variable
X := o1index; -- global variable
driver := A; -- driver = CAS

WHILE A > xst[X]
DO X := X + 1 OD;

WHILE A < xst[X]
DO X := X - 1 OD;

o1index := X; -- update o1index

Y := X + 1; -- Y := o1index + 1
A := xst[Y]; -- xst[x+1]
A := A - xst[X]; -- xst[x+1] - xst[x]
X := xst[Y]; -- xst[x+1]
X := X - driver; -- xst[x+1] - CAS
CALL sdivide; -- X/A ; X < A
-- ratio = (xst[x+1] - CAS)/(xst[x+1]-xst[x])

X := o1index; -- restore index in X
Y := X + 1;
X := fst[X];
X := X SUB fst[Y]; -- fst[x] - fst[x+1]
-- (fst[x] - fst[x+1]) is allways positive
-- An unsignned operation, SUB, is used since
-- fst[1]=fst[2]=1 is represented as '8000' which
-- is equivalent to -2147483648
-- -2147483648 minus any number cause a neg overflow
CALL smult; -- ratio*(fst[x] - fst[x+1])
X := o1index;
Y := X + 1;
A := A + fst[Y] -- A := KPSDOT = ratio*(fst[x]-fst[x+1]) + fst[x+1]
```

```
END; -- end oned


PROC folag: -- first order lag; scale of output is 4 times scale of input
-- (inputs : A,X,etau[X],ometau4[X],reset=var[3],state[X])
-- (outputs: A,state[X])
-- (calls: smult)
-- (destroys: A,X,Y)
-- time constants etau = e^-h/tau h:integration time step h=0.05 seconds
-- ometau4 = (1-etau)*4 (one minus etau times 4)
-- previous state in global state state[X]
-- global reset; X selects which and type of integrator
-- integrator input in A; output in A
BEGIN
INT etau[3], ometau4[3]; -- integrators time constant
INT lvselect, lv1, input; -- local variables
IF (Y := var[3]) Y /= 0 -- if reset then assign constant values, scale
-- input to state and output,
THEN  -- and make output = input

lvselect := X;
X := 2115480017; -- e^-0.05/3.33      scale 2^31
etau[1] := X; -- 0.985097148  ||
X := 2146141889; -- e^-0.05/80 ||
etau[2] := X; -- 0.999375195 ||
X := 128014524; -- (1-etau)*4 ||
ometau4[1] :=X; -- 0.059611408 ||
X := 5367034; -- (1-etau)*4 ||
ometau4[2] := X; -- 0.002499220 \/
A := A ADD 0; -- clear B
A := A<<1;
A := A ADD 0; -- clear B
A := A<<1; -- A := A*4
X := lvselect;

state[X] := A -- update state and return from reset with
-- output = input

ELSE -- no reset
lvselect := X; -- save select value
X := ometau4[X]; -- X := (1- etau)*4 scale 1 = 2^31 (bit32 -> 1)
CALL smult; -- scaled multiplication; factors in A and X
lv1 := A; -- save product [(1-etau)*4]*INPUT
X := lvselect;  -- get integrator select value
```

```
A := state[X]; -- get previous state of this integrator
X := etau[X];
CALL smult; -- etau*state
A := A + lv1; -- etau*state + (1-etau)*4*INPUT -> OUTPUT
X := lvselect;
state[X] := A -- update integrator state
FI
END; -- end of folag


PROC wout:
BEGIN
INT lin,nsmi; -- local variables
INT etau = 1513835370; -- e^-0.05/.143 = 0.704934527 scale 2^31
INT tau = 1228360647; -- 0.143 scale 2^33
IF ( Y := var[3]) Y /= 0 -- if rest then
THEN
state[3] := A; -- make state = input
A := 0 -- make output = 0 and return
ELSE
X := state[3]; -- get previous state scale 2^27
lin := A; -- put input in lin scale 2^27
A := X - lin; -- pstate - in
X := etau;
CALL smult; -- etau*(pstate-in)
nsmi := A; -- new state minus input
A := A + lin; -- newstate = etau*(pstate - in) + in
state[3] := A;
Y := 0;
X := Y - nsmi; -- in - state = in - (etau*(pstate-in)+in)
--         = -(etau*(pstate-in)
--         = -nsmi
A := tau; -- 0.143 scale 1 = 2^33
CALL sdivide; -- A := wout = (in - state)/tau scale 2^27
Y := Y ADD 0; A:= A<<1;
Y := Y ADD 0; A:= A<<1;
Y := Y ADD 0; A:= A<<1 -- scale to 1 = 2^30
FI

END; -- end wout


PROC smult: -- scaled multiplication is equivalent to long multiplication
-- (32 bits operand, 63 bits result) but the lower 31 bits
```

```
-- are truncated and discarded. The product of A*X will be
-- delivered in A shifted 31 times to the right.
-- If the product is negative the 2's complement is obtaine
-- after truncation. The factor X is always taken as positive
-- and a "1" in bit 31 (MSB) is not interpreted as a 2's
-- complement negative number.
BEGIN
INT facta,sign,product;
sign := A; -- make sign positive if A pos.; negative if A neg.
IF A < 0 THEN
facta := A;
Y := 0;
A := Y - facta -- make A := ABS(A) [absolute value]
FI;
facta := A;
A := 0;
Y := 32;
WHILE Y /= 0 -- test Y = 0 and jump if true. if Y /= 0 B is false
-- B always = 0 at beginning of DO
DO  A := A >> 1; -- shift right. first shift has no effect since A = 0
-- on arrival. this will result in shifting partial
-- product(A) only 31 time right while X is shifted
-- 32 times.
A := A ADD 0; -- clear B. This instruction is not necesary
X := X >> 1; -- put B (zero) in MSB; put LSB in B
IF B
THEN
A := A ADD facta;
IF B THEN A := 2; CALL error FI
FI;
Y := Y - 1 -- decrement counter

OD;
Y := sign;
IF Y >= 0 THEN RETURN -- A was positive, return positive

  ELSE  product := A; -- A was negative, make product neg.
Y := 0;
A := Y - product
FI
END; -- end smult


PROC sdivide:
```

```
-- performs X/A*2^31 result in A. if X>=A result = 2^31-1. if A = 0
-- result = 2^31-1

BEGIN
INT divtemp,div,sign; -- local variables
Y := 1;
sign := Y; -- make sign positive
IF A > 0 AND X < 0
THEN
sign := X; -- make sign negative
Y := 0;
X := Y - sign -- X := ABS(X)
ELSE
IF A < 0 AND X > 0
THEN
sign := A; -- make sign negtive
Y := 0;
A := Y - sign -- A := ABS(A)
FI
FI;
divtemp := A;
A := 0;
Y := 30;
A := A ADD 0; -- clear B
REPEAT A := A << 1;
X := X << 1;
IF X GE divtemp
THEN
X := X SUB divtemp;
A := A + 1
FI
UNTIL (Y := Y SUB 1) B;
Y := sign;
IF Y >= 0 THEN RETURN
  ELSE div := A;
Y := 0;
A := Y - div -- make quotient negative
FI
END;
```

```
PROC error:
BEGIN
STOP
END -- end error



FINISH
```

Appendix B: FORTRAN Yaw Damper Program

```
      SUBROUTINE YAWDAMP(DELRYD)
C
C**** THIS ROUTINE IS THE YAW CONTROL SYSTEM
C

C* CAS       CALIBRATED AIRSPEED                                KNOTS
C* DELRYD    RUDDER DEFLECTION DUE TO YAW DAMPER                DEG
C* RBDEG     YAW   RATE, BODY AXIS                              DEG/SEC
C* RESET     IS RESET MODE FLAG                                 LOGICAL
C* T         TIME                                               SECONDS

      INCLUDE 'CAB2.COM'
      INCLUDE 'CA737.COM'
      INCLUDE 'CA737IP.COM'
      INCLUDE 'CDEFLEC.COM'
      INCLUDE 'CEQMOTN.COM'
C
      REAL KPSIT,KPSDOT
      DIMENSION VCAST(8), KPSIT(8)
C
      DATA VCAST /-1.E30, 100., 122.4, 150.8, 206., 292., 450., 1.E30/
      DATA KPSIT /    1.,   1.,  .765,   .61,   .5, .395,  .31,  .31 /
C
      DATA IX/1/
      DATA YS5/0./
      DATA SCRYCS1, SCRYCS2 / 2 * 0. /
      DATA SCRYCS3, SCRYCS4 / 2 * 0. /
      DATA SCRYCS5, SCRYCS6 / 2 * 0. /
C
      XLIM(P, Q, R) = AMIN1(R, AMAX1(P, Q))
C
      KPSDOT = ONED(CAS, IX, VCAST, KPSIT)
C
      YS1 = RBDEG * KPSDOT * .84
C
      YS2 = FOLAG(YS1, 3.33, RESET, SCRYCS1, SCRYCS2)
C
      YS3 = WOUT(YS2, .143, RESET, SCRYCS5, SCRYCS6)
C
```

```
      YS6= YS5*.955
C
      YS7 = FOLAG(YS6, 80., RESET, SCRYCS3, SCRYCS4)
C
      IF (T .EQ. 0.) YS7=YS3
C
      YS5 = (YS3-YS7)*9.15
C
      YS5 = XLIM(YS5,-4.,4.)
C
      DELRYD=YS5
C

      RETURN
      END

      FUNCTION ONED(X, IXST, XST, FST)
C
C**** THIS FUNCTION PERFORMS A 1-DIMENSIONAL LOOK-UP
C
C**** X    IS THE DRIVER
C**** IXST IS THE POINTER
C**** XST  IS THE ARRAY OF BREAK POINTS FOR X
C**** FST  IS THE ARRAY OF DATA POINTS
C
      DIMENSION XST(1), FST(1)
C
      IF (IXST .LE. 0) IXST = 1
      GOTO 3
C
    1 IXST = IXST - 2
C
    2 IXST = IXST + 1
C
    3 DELX = X - XST(IXST + 1)
C
      IF (DELX .GT. 0.) GOTO 2
C
      DELX = X - XST(IXST)
C
      IF (DELX .LT. 0.) GOTO 1
C
      DELX = DELX / (XST(IXST+1) - XST(IXST))
C
```

```
      ONED = FST(IXST) + DELX * (FST(IXST+1) - FST(IXST))
C
      RETURN
      END

      FUNCTION FOLAG(IN, TAU, RESET, STATE, ETAU)
C
C**** THIS FUNCTION INTEGRATES A FIRST-ORDER DIFFERENTIAL
C****     EQUATION DESCRIBED BY THE TRANSFER FUNCTION:
C
C****          OUT          1
C****          ---   =  ---------
C****          IN       TAU S + 1
C
C**** USE:  OUT = FOLAG(IN, TAU, RESET, STATE, ETAU)
C
C**** OUT      THE ANSWER - LAGGED IN
C**** IN       INPUT SIGNAL - VARIABLE TO BE LAGGED
C**** TAU      TIME CONSTANT
C**** RESET    .TRUE. IF WANT TO HAVE NO LAG OR WANT TO I.C.
C**** STATE    PAST VALUE OF THE LAGGED OUTPUT
C**** ETAU     SAVED VALUE OF (E TO THE -H/TAU)
C
      INCLUDE 'CAB2.COM'
C
      LOGICAL RESET
C
      REAL IN
C
C**** CONSTANT TAU, SERIAL INTEGRATION
C
      IF (.NOT. RESET) GOTO 2
C
C**** RESET MODE, NO DELAY (OUT = IN) AND CALCULATE E TO THE -H/TAU
C
      STATE  = IN
      FOLAG  = STATE
      ETAU   = EXP(-H/TAU)
      RETURN
C
C**** OPERATE MODE, 1ST ORDER LAG IN EFFECT
C
    2 STATE = ETAU *(STATE - IN) + IN
      FOLAG = STATE
```

```
C
      RETURN
      END

      FUNCTION WOUT(IN, TAU, RESET, STATE, ETAU)
C
C**** THIS FUNCTION INTEGRATES A FIRST-ORDER DIFFERENTIAL
C****     EQUATION DESCRIBED BY THE TRANSFER FUNCTION:
C
C****          OUT          S
C****          ---   =  ---------
C****          IN       TAU S + 1
C
C**** USE:  OUT = WOUT(IN, TAU, RESET, STATE, ETAU)
C
C**** OUT      THE ANSWER - WASHED OUT IN
C**** IN       INPUT SIGNAL - VARIABLE TO BE WASHED OUT
C**** TAU      TIME CONSTANT
C**** RESET    .TRUE. IF WANT TO HAVE NO WASH OUT OR WANT TO I.C.
C**** STATE    PAST VALUE OF THE WASHED OUT OUTPUT
C**** ETAU     SAVED VALUE OF (E TO THE -H/TAU)
C
      INCLUDE 'CAB2.COM'
C
      REAL IN
C
      LOGICAL RESET
C
C**** CONSTANT TAU, SERIAL INTEGRATION
C
   10 IF (.NOT. RESET) GOTO 2
C
C**** RESET MODE, SET OUTPUT TO ZERO AND SAVE PAST VALUE AND
C****     THE EXPONENTIAL
C
      WOUT  = 0.
      STATE = IN
      ETAU  = EXP(-H/TAU)
      RETURN
C
C**** OPERATE MODE, COMPUTE THE NEW OUTPUT
C
    2 STATE = ETAU*(STATE - IN) + IN
      WOUT  = (IN - STATE) / TAU
```

```
C
      RETURN
      END
```

Table 1. VIPER function field

FF =			
	0		D := ~M -- Load M into D complemented
	1		Y := P then P := M -- Jump to M storing P in Y
	2		D := Input -- Load input into D } Equivalent if
	3		D := M -- Load M into D } load inmediate
	4		D := R + M, B := carry
	5		D := R + M, stop on overflow
	6		D := R - M, B := borrow
	7		D := R - M, stop on overflow
	8		D := R XOR M
	9		D := R AND M
	10		D := R NOR M
	11		D := R AND ~M
	12	MF=0	D := R/2, sign bit copied -- shift right
		MF=1	D := R>>1 through B -- circular shift right through B
		MF=2	D := R*2, stop on overflow -- shift left
		MF=3	D := R<<1 through B -- circular shift left through B
	13		STOP
	14		STOP
	15		STOP

Table 2. Scale Factors

CAS	0 .. +1000 Knots	4.7683 x 10 Knots/MU*
	/1024	2,097,152 MU/Knots
KPSDOT	0.31 .. 1 RU*	4.6566 x 10 RU/MU
	/ 1	2,147,483,648 MU/RU
RBDEG	-20 .. +20 Deg/Second	1.4901 x 10 Deg/Sec/MU
	/32	67,108,864 MU/RU
VYS1	-20 .. +20 RU	1.4901 x 10 RU/MU
	/32	67,108,864 MU/RU
YS1	-16 .. +16 RU	7.4506 x 10 RU/MU
	/16	134,217,728 MU/RU
YS2	-3 .. +3 RU	1.8626 x 10 RU/MU
	/ 4	536,870,912 MU/RU
YS3	-2 .. +2 RU	9.3132 x 10 RU/MU
	/ 2	1,073,741,824 MU/RU
YS5P1	-1 .. +1	4.6566 x 10 RU/MU
	/ 1	2,147,483,648 MU/RU
YS5P2	-10 .. +10	7.4506 x 10 RU/MU
	/16	134,217,728 MU/RU
YS5	-4 .. +4	1.8626 x 10 RU/MU
	/ 4	536,870,912 MU/RU
YS6	-3.82 .. +3.82	1.8626 x 10 RU/MU
	/ 4	536,870,912 MU/RU
YS7	-1 .. +1	4.6566 x 10 RU/MU
	/ 1	2,147,483,648 MU/RU

* MU - Machine Units
RU - Real Units

Table 3. Example Run

Case Number	1
Wind Direction	109.3300 degrees
Wind Speed	5.604287 knots
Wind Type	- 1
Gust Amplitud	6.305861 fps
Gusts	on
Max. Relative VIPER Yaw Damper Error	4.1937530×10^{-2}
Max. Relative FORTRAN YD Error	7.0957579×10^{-2}
Max. Absolute VIPER Yaw damper Error	1.2159347×10^{-6}
Max. Absolute FORTRAN YD Error	1.2509711×10^{-5}
Average Relative VIPER YD Error	3.8307622×10^{-5}
Average Relative FORTRAN YD Error	9.0578076×10^{-5}
Average Absolute VIPER YD Error	4.3536136×10^{-7}
Average Absolute FORTRAN YD Error	1.2325295×10^{-6}

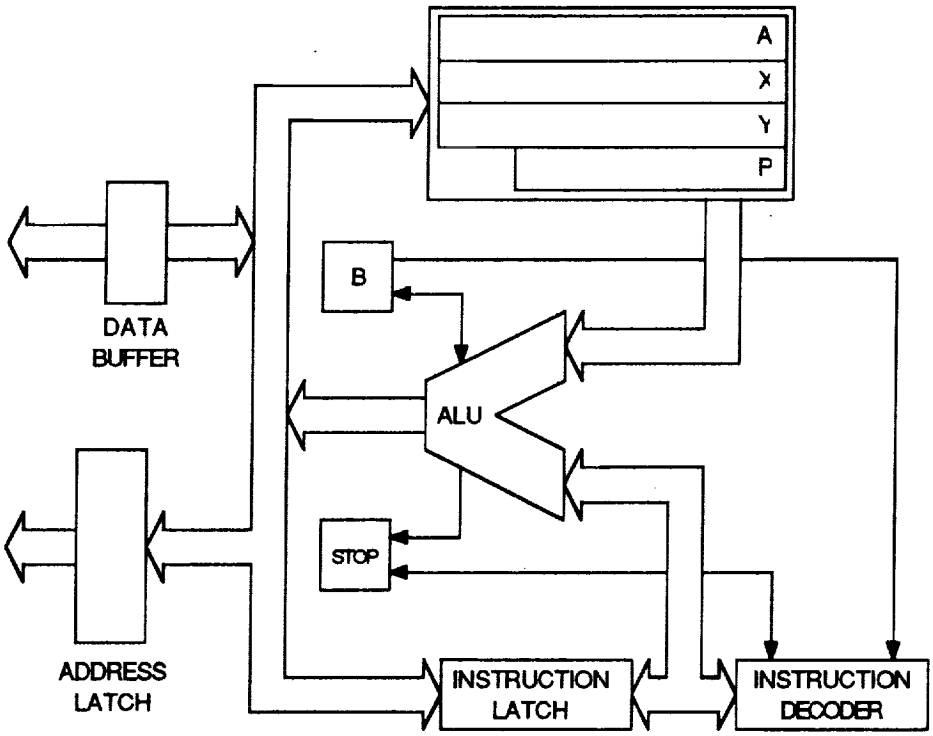


Figure 1. Viper Architecture

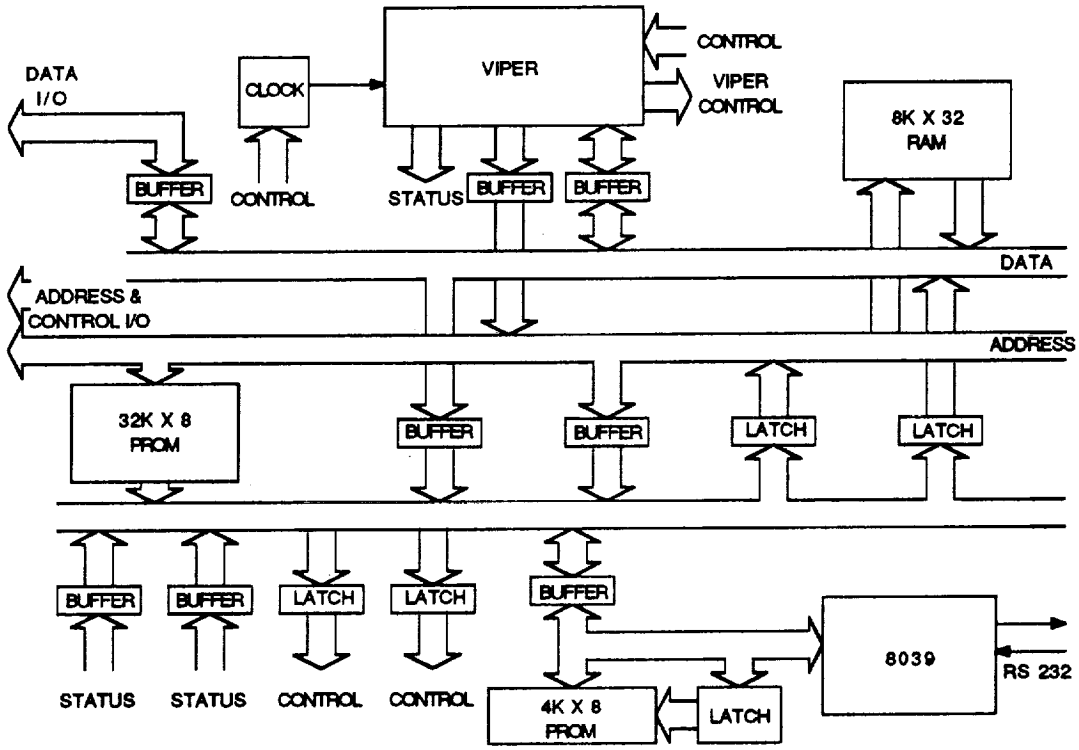


Figure 2. VIPER Single Board Computer

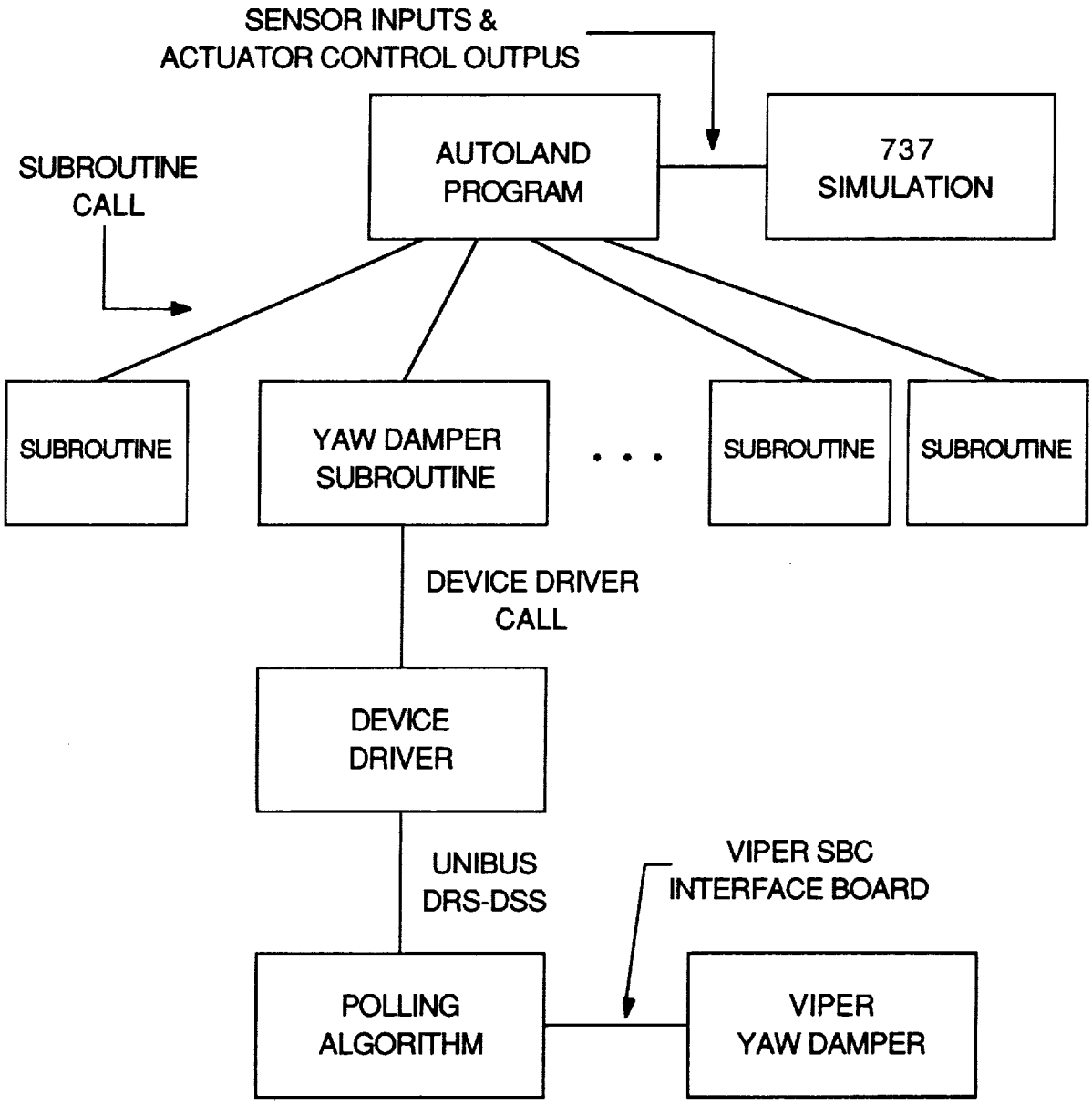


Figure 3. Software Organization

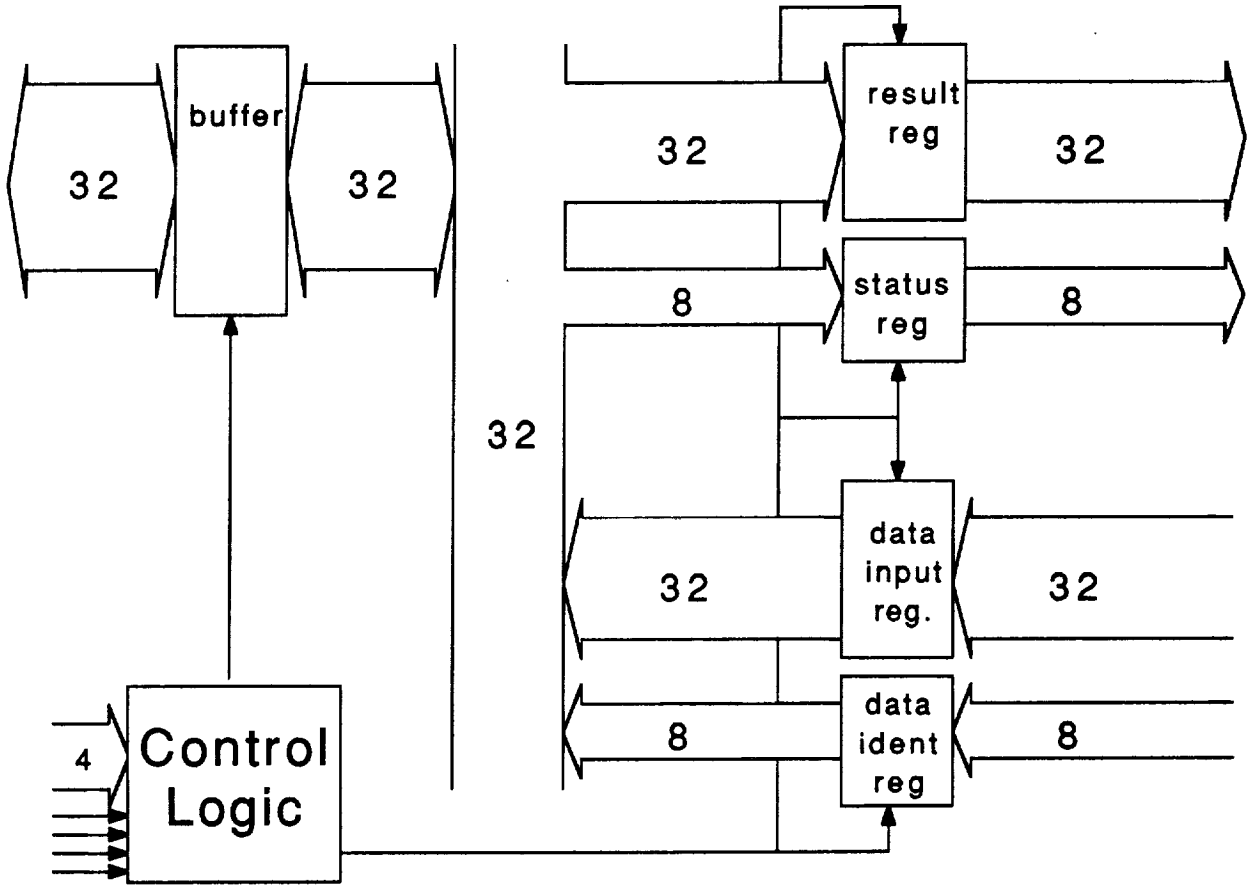
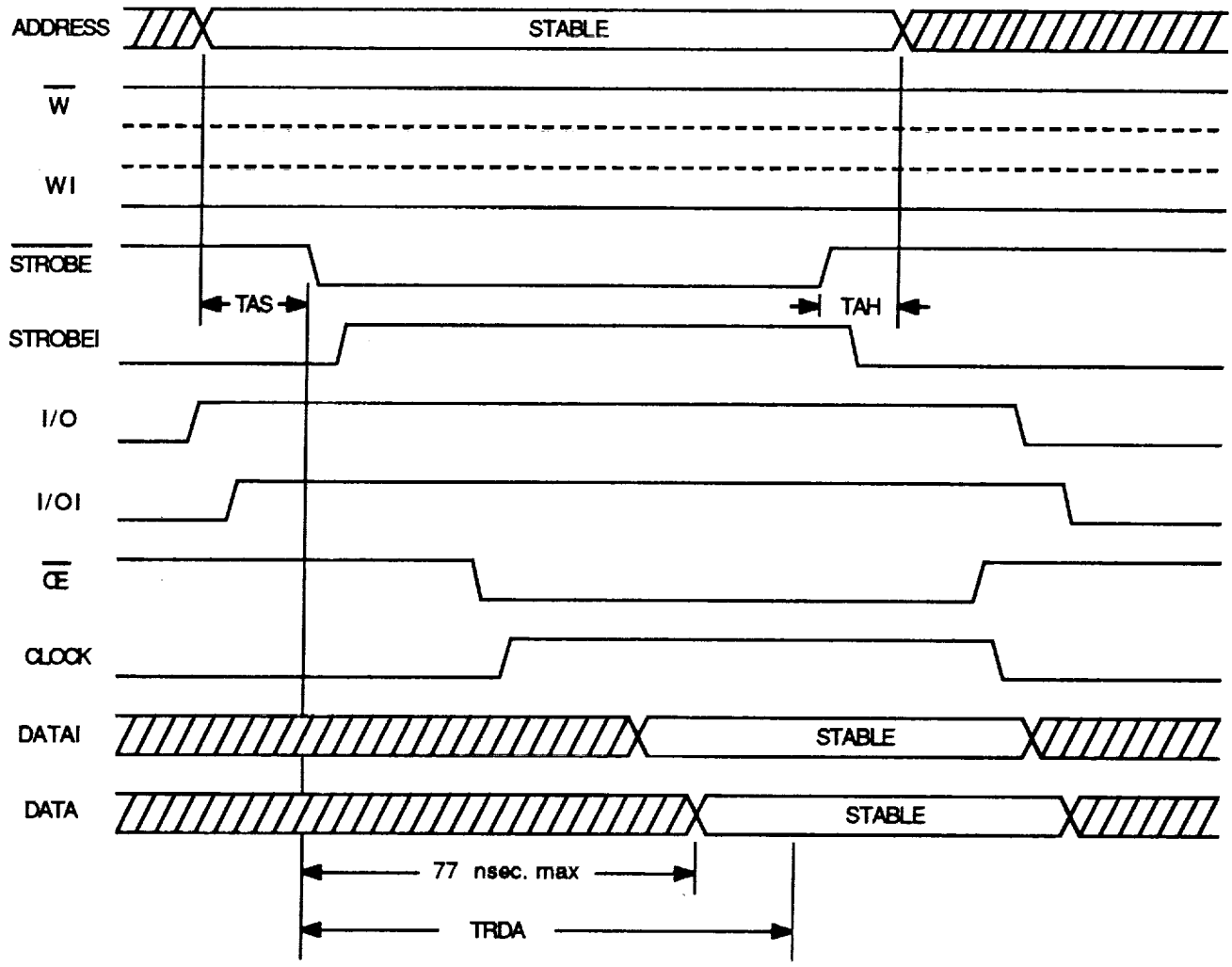
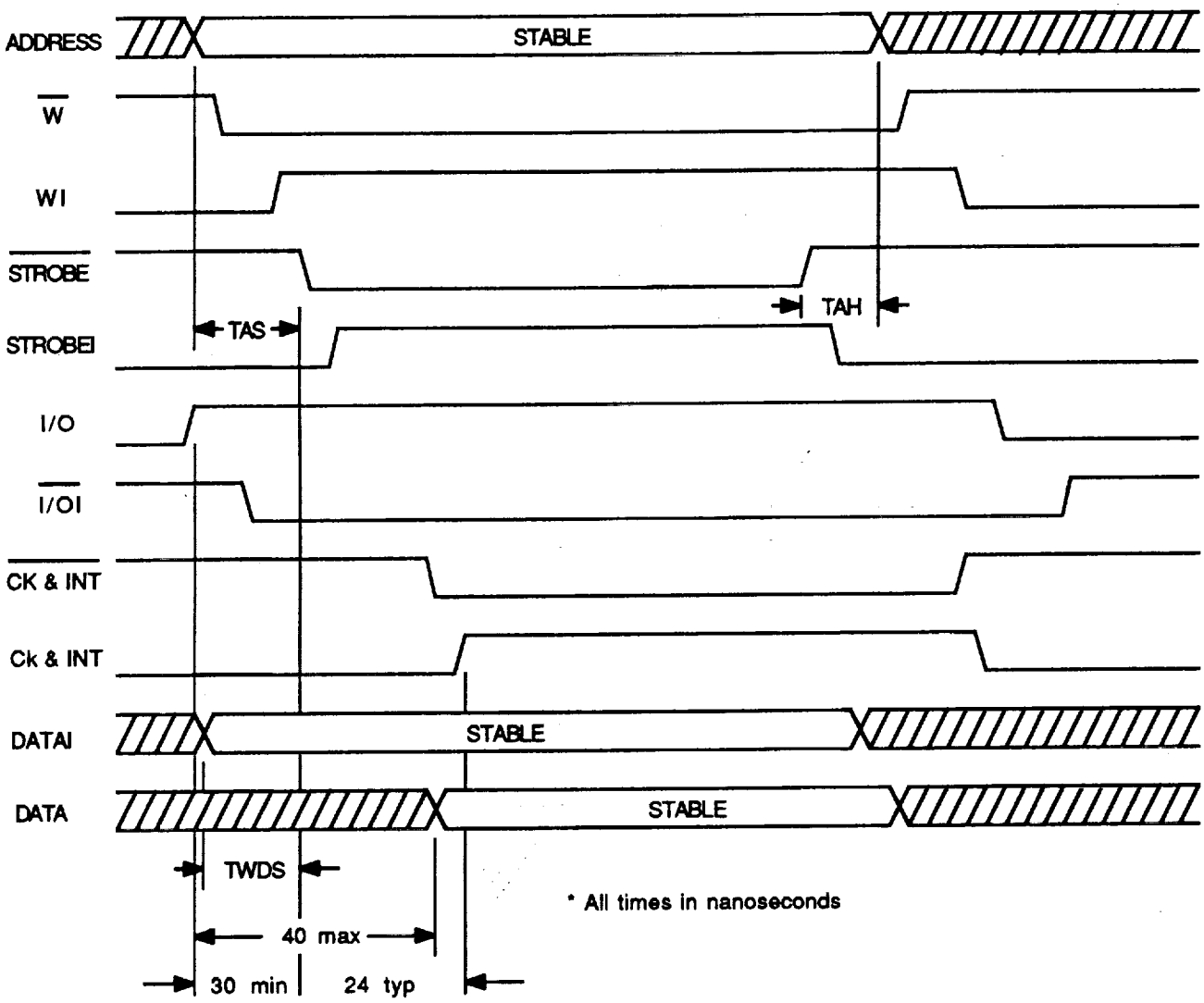


Figure 4. Interface Board



TAS: Address Set-up	30 nseconds minimum
TAH: Address Hold	20 nseconds minimum
TRDA: Read Dta Available	150 nseconds maximum

Figure 5a. Interface Board Read Cycle



TAS: Address Set-up	30 nseconds minimum
TAH: Address Hold	20 nseconds minimum
TWDS: Write Data Set-up	20 nseconds minimum

Figure 5b. Interface Board Write Cycle

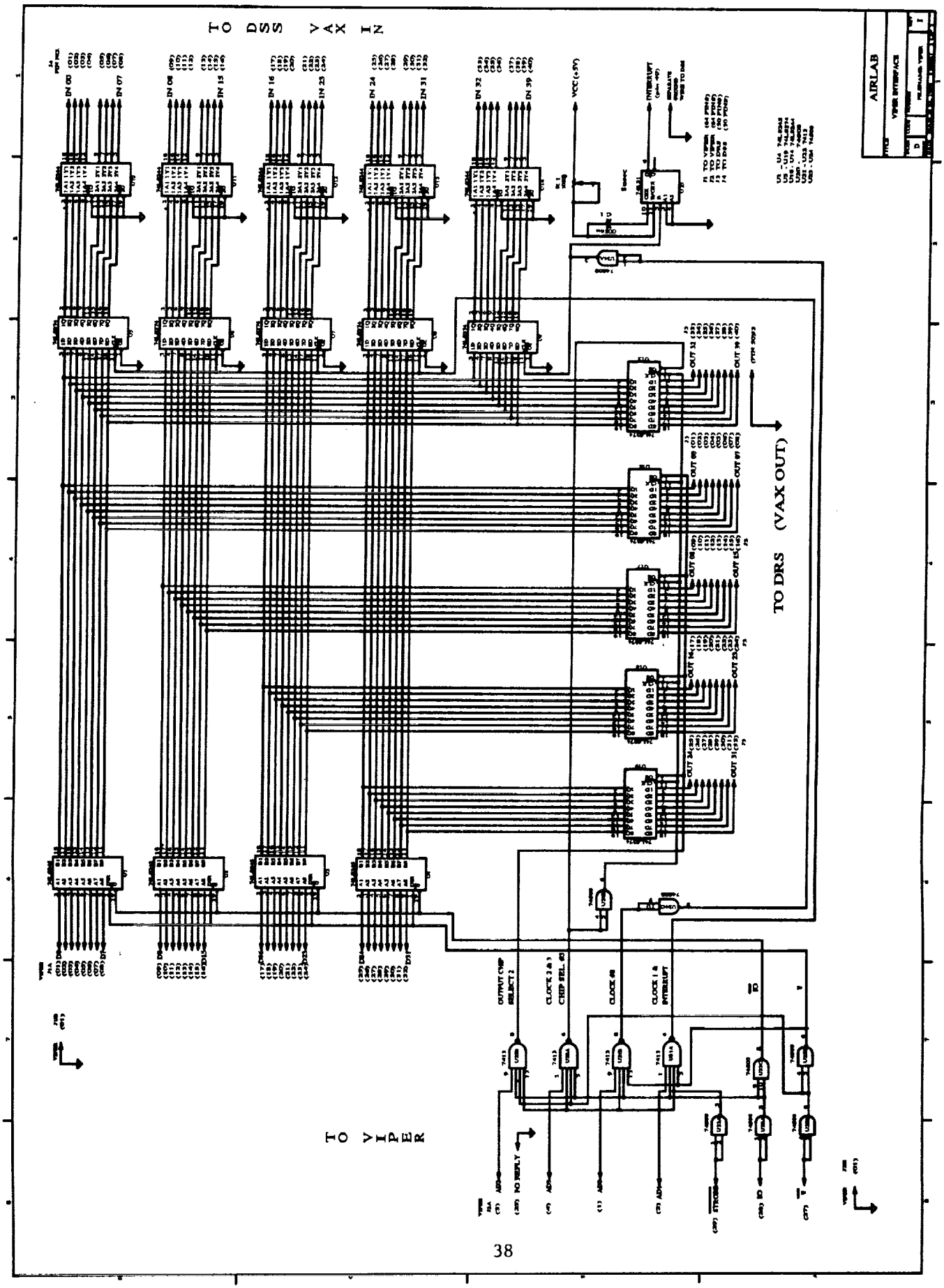


Figure 6. Interface Board Schematic

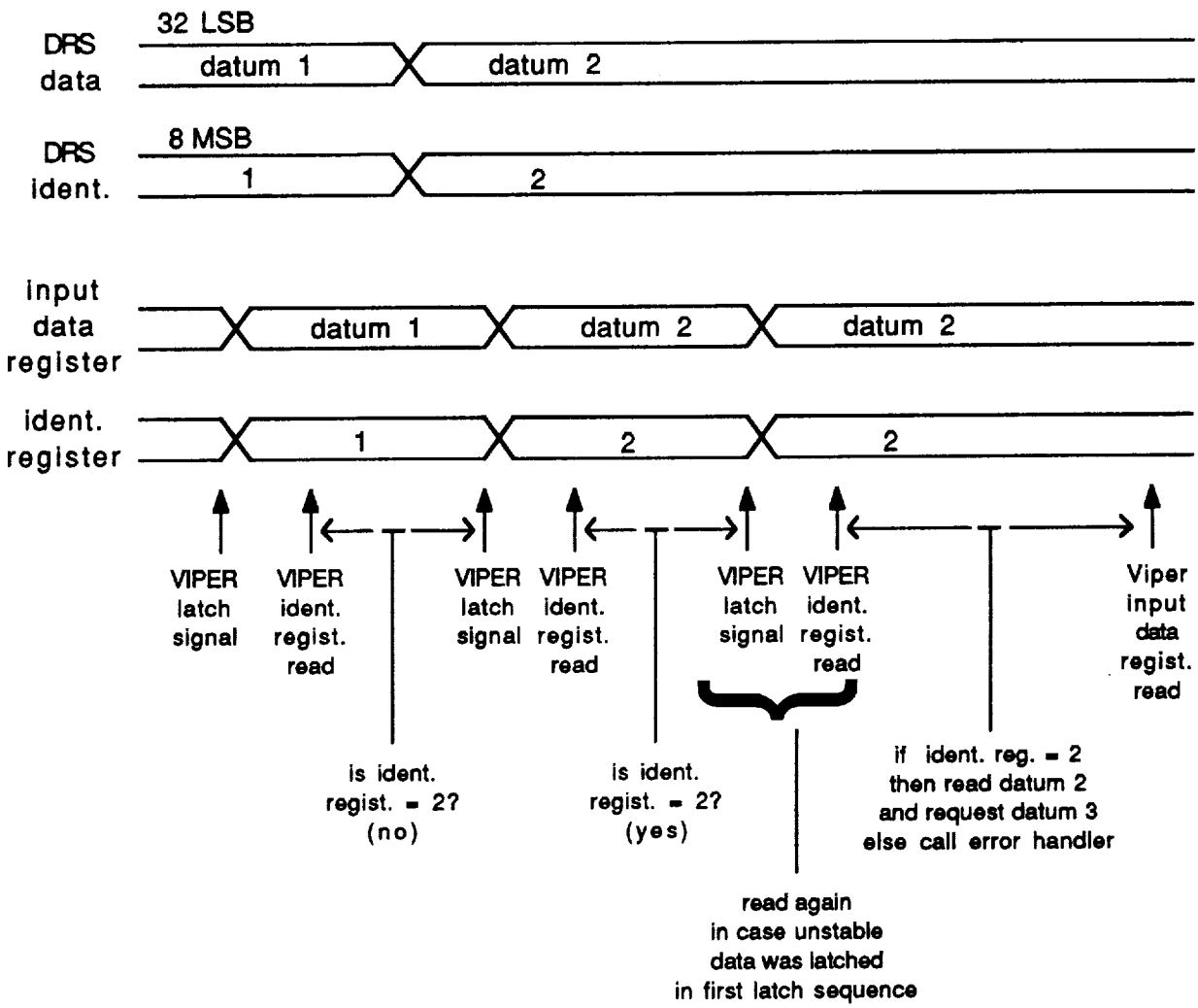


Figure 7a. Polling Time Diagram

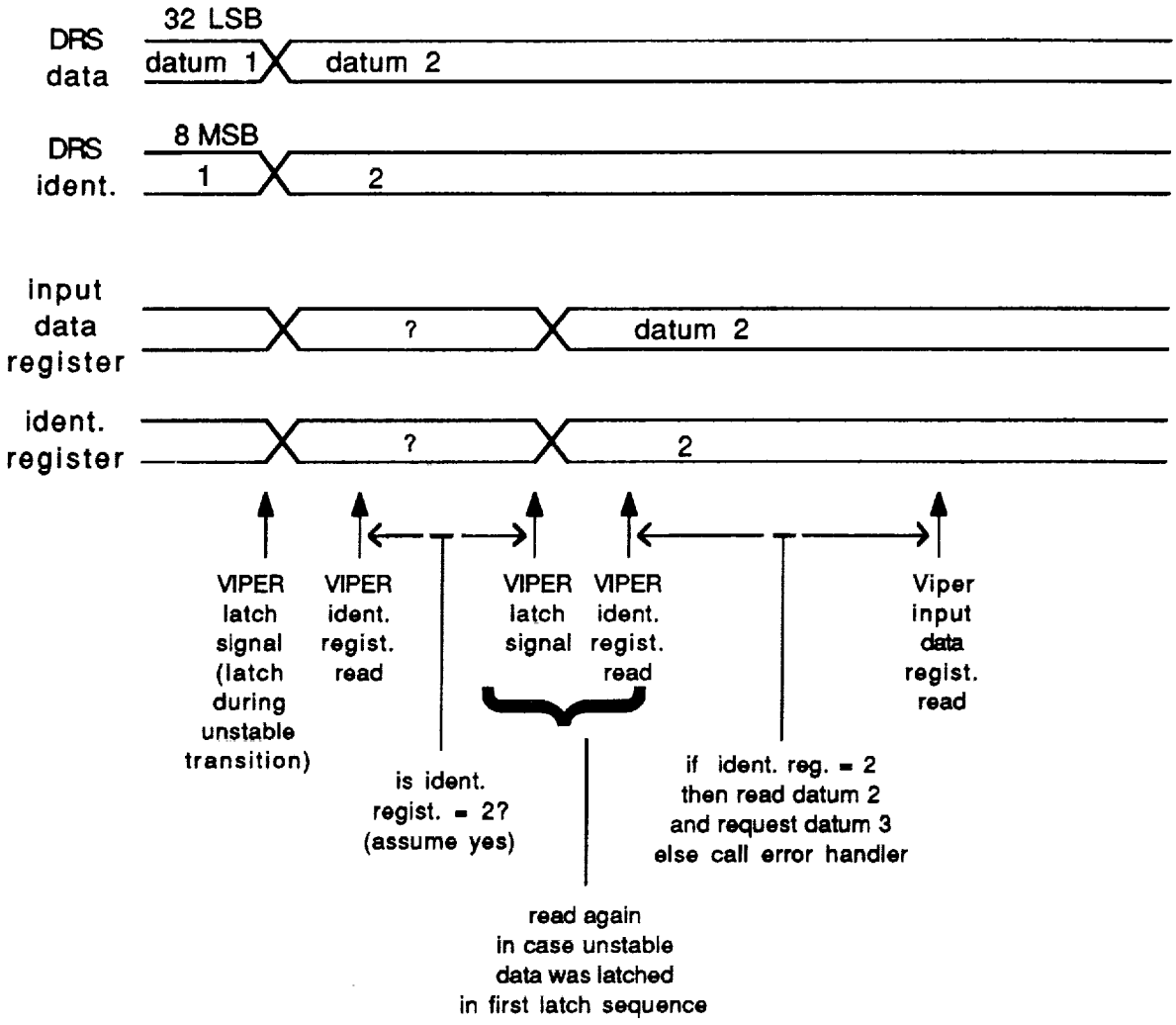


Figure 7b. Polling Time Diagram

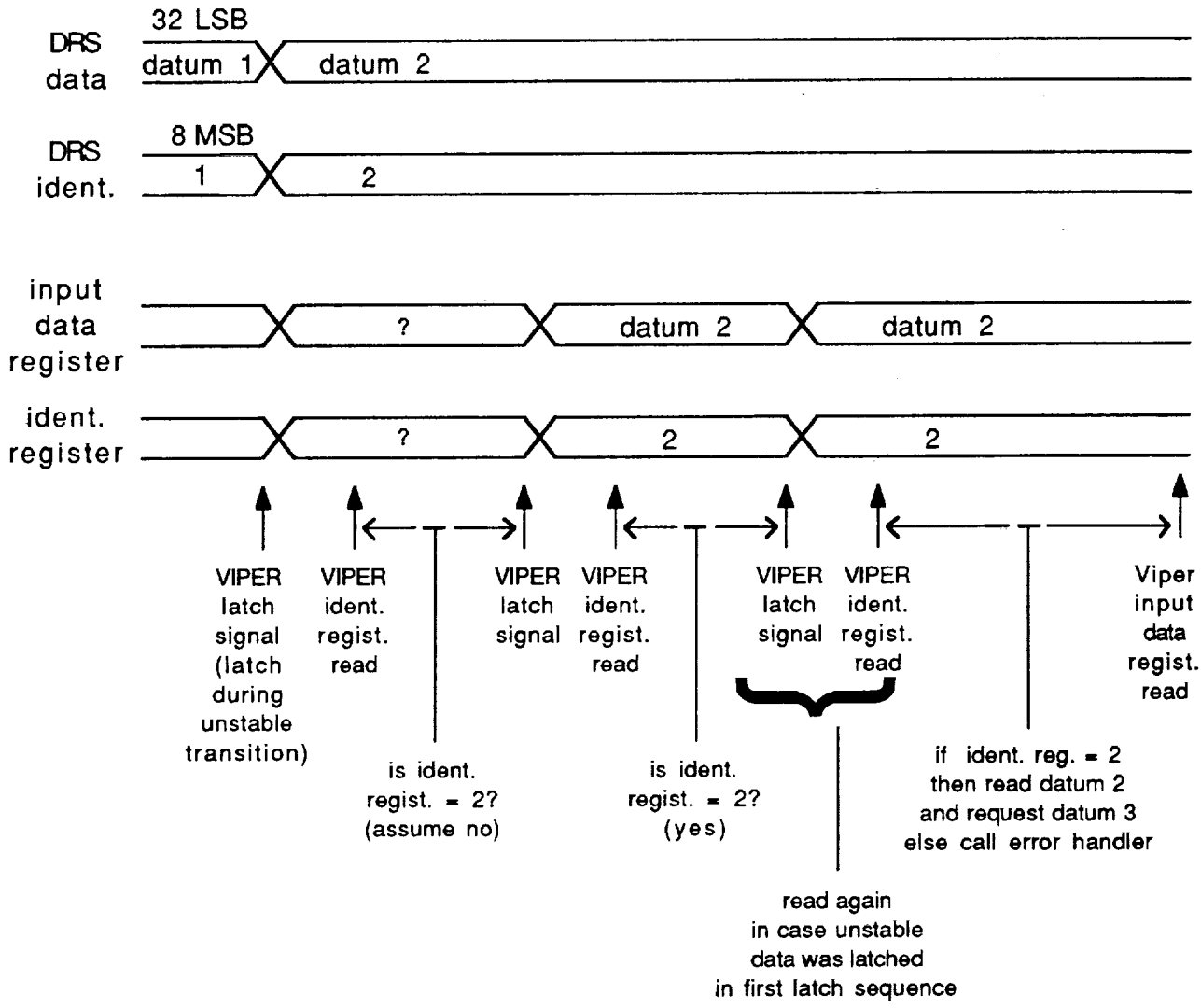


Figure 7c. Polling Time Diagram

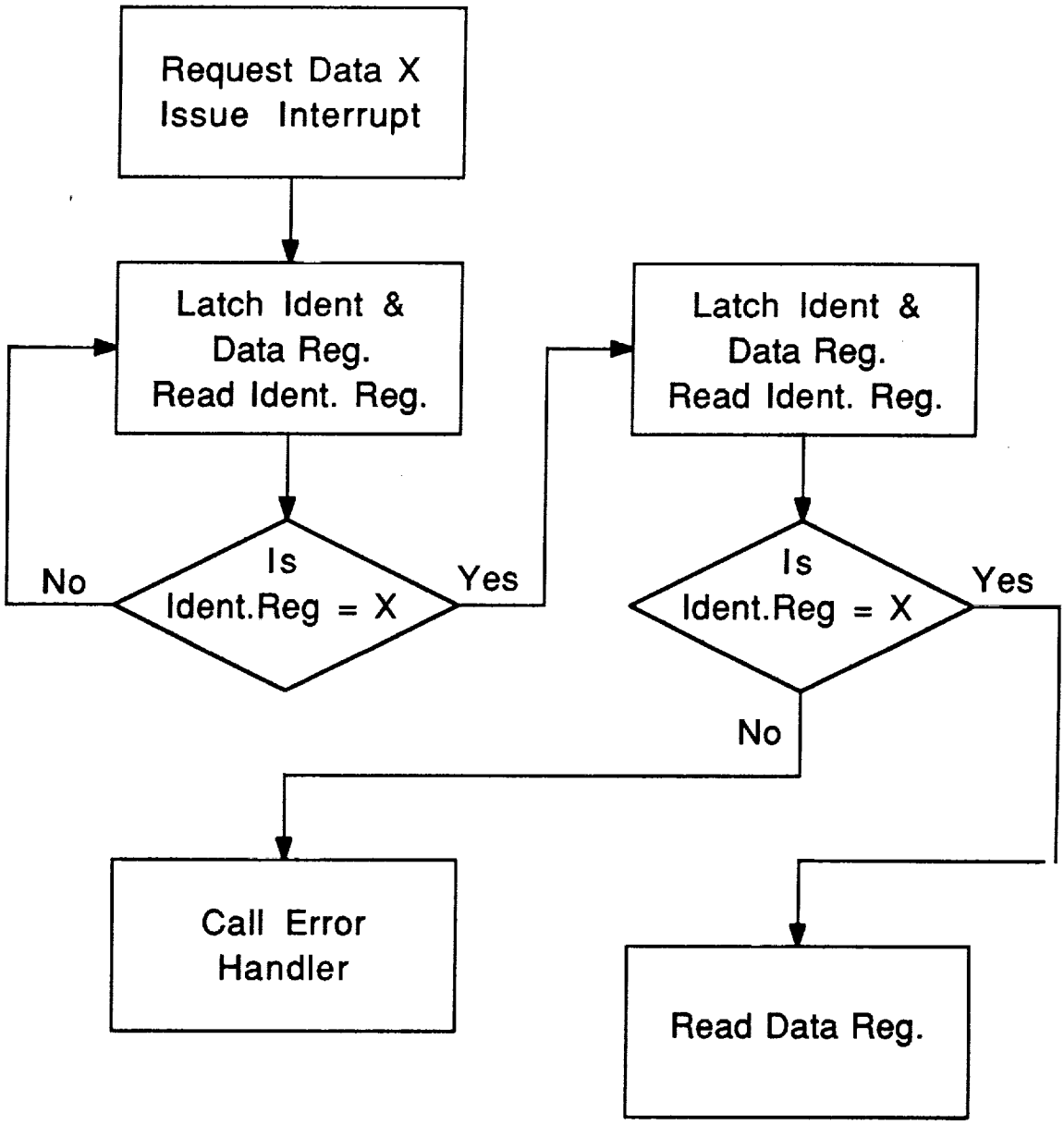


Figure 8. Polling Algorithm

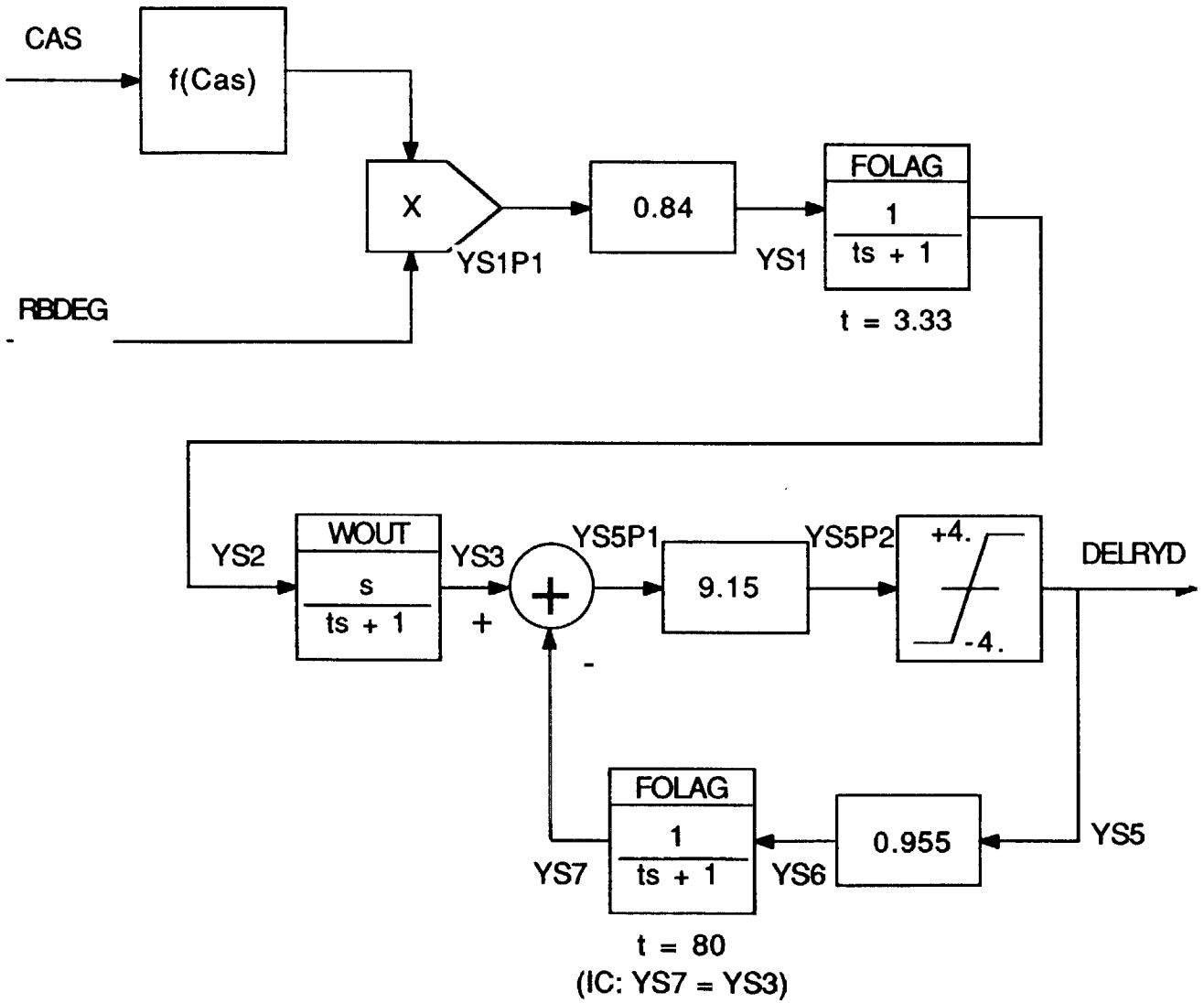


Figure 9. Yaw Damper s-plane

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-104098		

4. Title and Subtitle	5. Report Date
A Case Study for the Real-Time Experimental Evaluation of the VIPER Microprocessor	September 1991
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Victor A. Carreno Rob K. Angellatta	
	10. Work Unit No.
	505-64-10-05

9. Performing Organization Name and Address	11. Contract or Grant No.
NASA Langley Research Center Hampton, VA 23665-5225	
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Technical Memorandum
National Aeronautics and Space Administration Washington, DC 20546	14. Sponsoring Agency Code

15. Supplementary Notes

Victor A. Carreno: Langley Research Center, Hampton, VA
Rob K. Angellatta: Lockheed Engineering and Sciences Company, Hampton, VA

16. Abstract

This paper describes an experiment to evaluate the applicability of the Verifiable Integrated Processor for Enhanced Reliability (VIPER) microprocessor to real-time control. The VIPER microprocessor was invented by the Royal Signals and Radar Establishment (RSRE), U.K., and is an example of the use of formal mathematical methods for developing electronic digital systems with a high degree of assurance on the system design and implementation correctness.

The experiment consisted of selecting a control law, writing the control law algorithm for the VIPER processor, and providing real-time, dynamic inputs to the processor and monitoring the outputs. The control law selected and coded for the VIPER processor was the yaw damper function of an automatic landing program for a 737 aircraft.

The mechanisms for interfacing the VIPER Single Board Computer to the VAX host are described in the paper. The yaw damper control law as well as the Vista programming language for the VIPER are also described. Results include run time experiences, performance evaluation, and comparison of VIPER and FORTRAN yaw damper algorithm output for accuracy estimation.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Formal Methods Design Correctness Real-Time Control Microprocessor Scaling Integer Arithmetic	Unclassified-Unlimited Subject Category 33

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	44	A03

NASA FORM 1626 OCT 86